

Blockchain Lab

Experiment 5

Sanket More
D20A 35

Aim: Deploying a Voting/Ballot Smart Contract

Theory

1. Relevance of required Statements in Solidity Programs

In Solidity, the `require` statement is used as a validation mechanism to enforce conditions before executing a function. It acts as a guard clause that ensures only valid inputs, authorized users, or correct states are allowed to proceed further in the program.

If the condition specified inside the `require` statement evaluates to `false`, the following actions occur:

- The function execution immediately stops.
- All state changes made during that transaction are reverted.
- The remaining gas is refunded (except the gas already consumed).
- An optional error message is returned.

This rollback mechanism is crucial in blockchain applications because it prevents invalid or malicious transactions from altering the contract's state permanently. Since blockchain data is immutable, such protective mechanisms are essential for maintaining integrity and security.

Example: Voting Smart Contract

In a Voting (Ballot) Smart Contract, `require` can be used in the following scenarios:

To check whether a voter has the right to vote:

```
require(voters[msg.sender].weight > 0, "Has no right to vote");
```

- To ensure that a voter has not already voted.
- To verify that only the chairperson can grant voting rights.

Therefore, `require` statements enhance:

- **Security** – Prevent unauthorized access.
- **Correctness** – Ensure logical conditions are satisfied.
- **Reliability** – Maintain consistency of blockchain data.
- **User Experience** – Provide meaningful error messages for debugging.

2. Important Keywords: `mapping`, `storage`, and `memory`

(a) `mapping`

A `mapping` in Solidity is a key-value data structure similar to a hash table or dictionary. Its syntax is:

```
mapping(keyType => valueType)
```

Example:

```
mapping(address => Voter) public voters;
```

In this case, each Ethereum address is associated with a `Voter` structure.

Characteristics of Mapping:

- Provides fast lookup of values using keys.
- Does not store keys explicitly.
- Does not have a length property.
- Cannot be iterated over directly.
- More gas-efficient for data retrieval compared to arrays.

Mappings are widely used in smart contracts such as voting systems, token balances, and ownership tracking.

(b) storage

storage refers to the permanent data area of a smart contract stored on the Ethereum blockchain. Variables declared at the contract level are stored in storage by default.

Key Features:

- Data is persistent across transactions.
- Changes remain recorded on the blockchain.
- Writing to storage consumes higher gas fees.
- Suitable for maintaining contract state.

For example, voter details stored inside a mapping remain available throughout the contract's lifecycle unless explicitly modified.

(c) memory

memory is a temporary data location used during function execution. Variables declared in memory exist only for the duration of the function call.

Key Features:

- Data is discarded after function execution.
- Less expensive than storage.
- Used for temporary variables, parameters, and intermediate computations.

Example use cases include:

- Temporary string manipulation.
- Handling function arguments.
- Processing proposal names before storing them permanently.

A smart contract developer must carefully decide whether to use **storage** or **memory** to optimize gas costs and ensure efficiency.

3. Why Use `bytes32` Instead of `String`?

In earlier implementations of Ballot smart contracts, `bytes32` was commonly used for storing proposal names instead of `string`. The primary reason for this choice was gas efficiency and performance optimization.

`bytes32`

- Fixed-size data type (exactly 32 bytes).
- Requires less storage management.
- Faster comparison operations.
- Lower gas consumption.
- Limited to 32 characters.

Due to its fixed size, `bytes32` is efficient but not flexible for longer or user-friendly names.

`string`

- Dynamically sized data type.
- Can store variable-length text.
- More user-friendly and readable.
- Requires complex memory handling in the Ethereum Virtual Machine (EVM).
- Higher gas consumption.

While `string` improves usability and flexibility, it increases computational and storage costs.

CODE:-

```
// SPDX-License-Identifier: GPL-3.0

//SANKET MORE D20A 35
pragma solidity >=0.7.0 <0.9.0;

contract Ballot {

    struct Voter {
        uint weight;
        bool voted;
        address delegate;
        uint vote;
    }

    // ✓ Changed bytes32 → string
    struct Proposal {
        string name;
        uint voteCount;
    }

    address public chairperson;

    mapping(address => Voter) public voters;

    Proposal[] public proposals;

    // ✓ Changed constructor parameter
    constructor(string[] memory proposalNames) {
        chairperson = msg.sender;
        voters[chairperson].weight = 1;

        for (uint i = 0; i < proposalNames.length; i++) {
            proposals.push(Proposal({
                name: proposalNames[i],
                voteCount: 0
            }));
        }
    }
}
```

```

function giveRightToVote(address voter) external {
    require(msg.sender == chairperson, "Only chairperson can give right to vote.");
    require(!voters[voter].voted, "The voter already voted.");
    require(voters[voter].weight == 0, "Voter already has the right to vote.");

    voters[voter].weight = 1;
}

function delegate(address to) external {
    Voter storage sender = voters[msg.sender];

    require(sender.weight != 0, "You have no right to vote");
    require(!sender.voted, "You already voted.");
    require(to != msg.sender, "Self-delegation is disallowed.");

    while (voters[to].delegate != address(0)) {
        to = voters[to].delegate;
        require(to != msg.sender, "Found loop in delegation.");
    }

    Voter storage delegate_ = voters[to];
    require(delegate_.weight >= 1);

    sender.voted = true;
    sender.delegate = to;

    if (delegate_.voted) {
        proposals[delegate_.vote].voteCount += sender.weight;
    } else {
        delegate_.weight += sender.weight;
    }
}

function vote(uint proposal) external {
    Voter storage sender = voters[msg.sender];

    require(sender.weight != 0, "Has no right to vote");
    require(!sender.voted, "Already voted.");
}

```

```

        sender.voted = true;
        sender.vote = proposal;

        proposals[proposal].voteCount += sender.weight;
    }

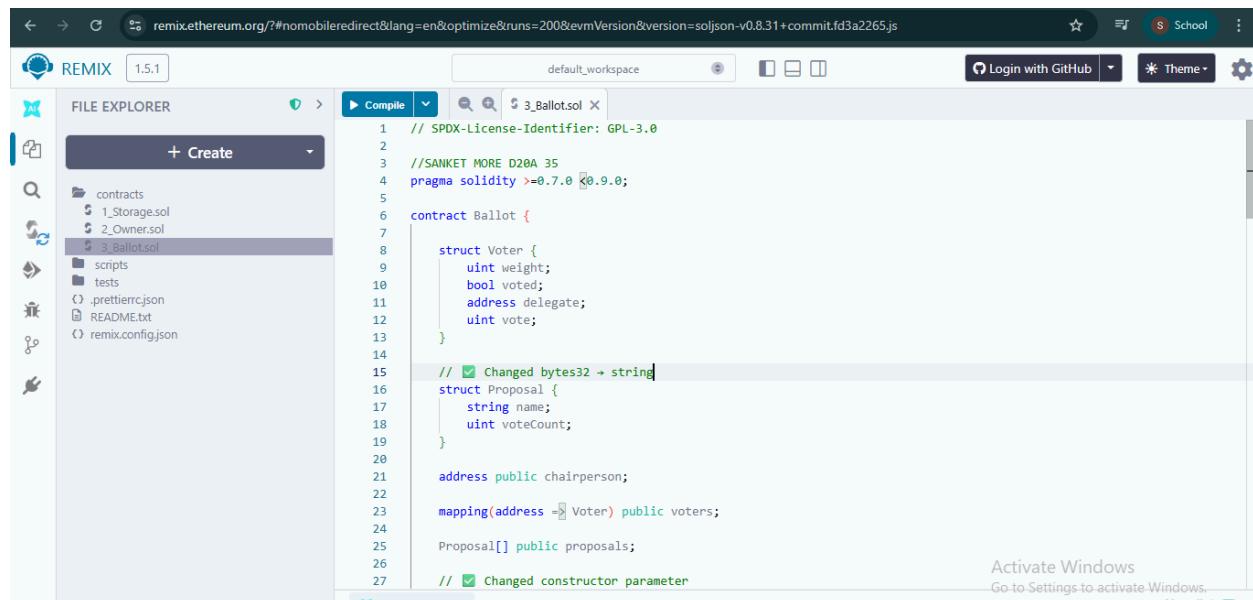
function winningProposal() public view returns (uint winningProposal_) {
    uint winningVoteCount = 0;

    for (uint p = 0; p < proposals.length; p++) {
        if (proposals[p].voteCount > winningVoteCount) {
            winningVoteCount = proposals[p].voteCount;
            winningProposal_ = p;
        }
    }
}

// ✅ Changed return type
function winnerName() external view returns (string memory winnerName_) {
    winnerName_ = proposals[winningProposal()].name;
}
}

```

OUTPUT:-



The screenshot shows the Remix IDE interface with the following details:

- File Explorer:** Shows contracts (1.Storage.sol, 2.Owner.sol, 3.Ballot.sol), scripts, tests, .prettiercjson, README.txt, and remix.config.json.
- Code Editor:** Displays the Solidity code for the Ballot contract. The code includes comments indicating changes to the constructor parameter and the bytes32 + string fields.
- Compiler:** Shows the compiler version as 1.5.1 and the selected file as 3_Ballot.sol.
- Toolbars and Buttons:** Includes standard browser navigation buttons, a search bar, and Remix-specific buttons like 'Compile' and 'Deploy'.
- Status Bar:** Shows the URL remix.ethereum.org/#nomobile&redirect&lang=en&optimize&runs=200&evmVersion&version=soljson-v0.8.31+commit.fd3a2265.js, a 'School' icon, and a GitHub integration button.
- Bottom Bar:** Includes links for 'Activate Windows' and 'Go to Settings to activate Windows.'

Deploying and running of the contract

The screenshot shows the Remix IDE interface. On the left, there's a sidebar with various icons for file operations like Open, Save, and Deploy. The main area has tabs for 'DEPLOY & RUN TRANSACTIONS' and 'COMPILER'. The 'DEPLOY & RUN TRANSACTIONS' tab is active, showing fields for 'ACCOUNT' (set to 0x5B3...eddC4), 'GAS LIMIT' (set to 'Estimated Gas'), 'VALUE' (set to 0 Wei), and a 'CONTRACT' dropdown set to 'Ballot - contracts/3_Ballot.sol'. Below these are buttons for 'Deploy' (with a dropdown for proposal names) and 'At Address'. The right side of the screen displays the Solidity code for the 'Ballot' contract, which includes definitions for a 'Voter' struct, a 'Proposal' struct, and the 'Ballot' contract itself with its constructor and other functions. A status bar at the bottom indicates 'Transactions recorded 0 i' and 'Deployed Contracts 0'. A message bar at the bottom says 'Did you know? You can use 'Generate documentation' in the right-click menu to get AI-generated documentation.'

```
// SPDX-License-Identifier: GPL-3.0
//SANKEY MORE D20A 35
pragma solidity >=0.7.0 <0.9.0;

contract Ballot {
    struct Voter {
        uint weight;
        bool voted;
        address delegate;
        uint vote;
    }

    struct Proposal {
        string name;
        uint voteCount;
    }

    address public chairperson;
    mapping(address => Voter) public voters;
    Proposal[] public proposals;
}

// Changed bytes32 -> string
```

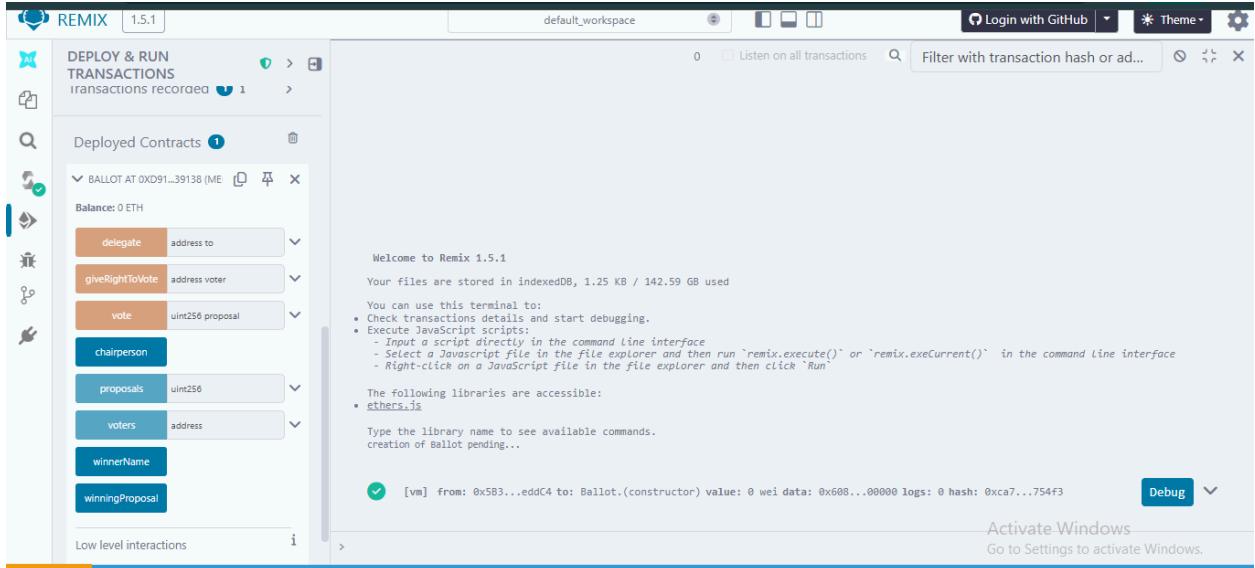
This screenshot shows the same Remix IDE interface after deployment. The 'DEPLOY & RUN TRANSACTIONS' tab is still active, but the 'Deploy' button now has a dropdown with the value '["Proposal1","Proposal2","Propc..."]'. The right side of the screen now displays a terminal-like interface with a welcome message and instructions for using the terminal. It also lists accessible libraries ('ethers.js') and provides a command to see available commands ('Type the library name to see available commands. creation of Ballot pending...'). At the bottom, a transaction log is shown: '[vm] from: 0x5B3...eddC4 to: Ballot.(constructor) value: 0 wei data: 0x608...00000 logs: 0 hash: 0xca7...754f3'. A 'Debug' button is visible at the bottom right. A message bar at the bottom says 'Activate Windows Go to Settings to activate Windows.'

```
Welcome to Remix 1.5.1
Your files are stored in indexedDB, 1.25 KB / 142.59 GB used
You can use this terminal to:
• Check transactions details and start debugging.
• Execute JavaScript scripts:
  - Input a script directly in the command line interface
  - Select a Javascript file in the file explorer and then run `remix.execute()` or `remix.executeCurrent()` in the command line interface
  - Right-click on a Javascript file in the file explorer and then click 'Run'

The following libraries are accessible:
• ethers.js

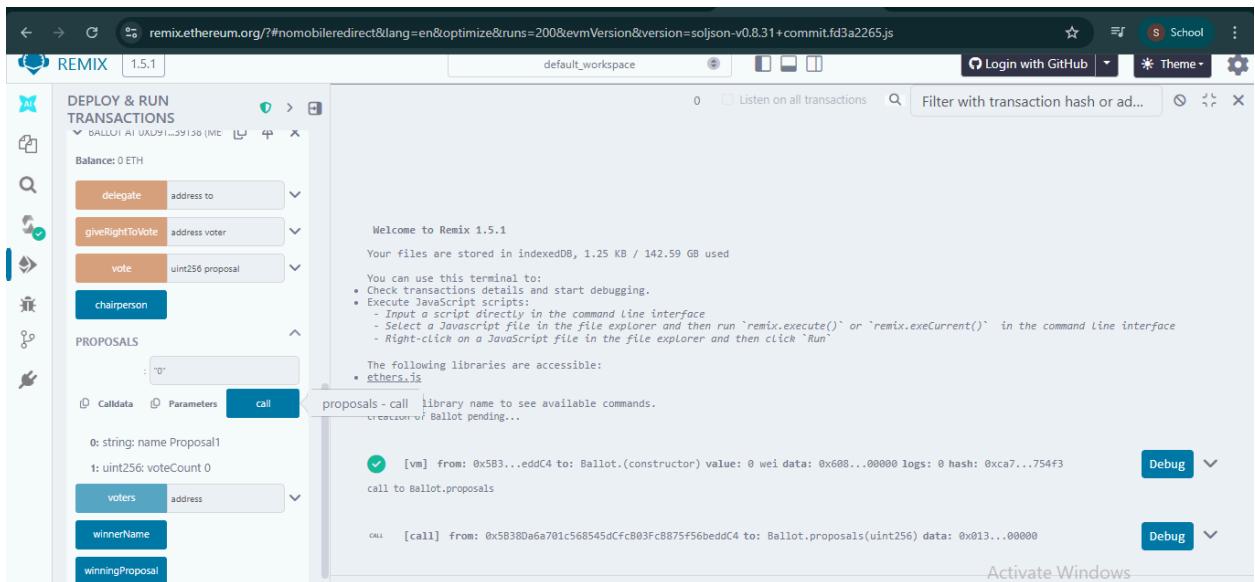
Type the library name to see available commands.
creation of Ballot pending...
```

Loading the Proposal Candidate's Names (string)



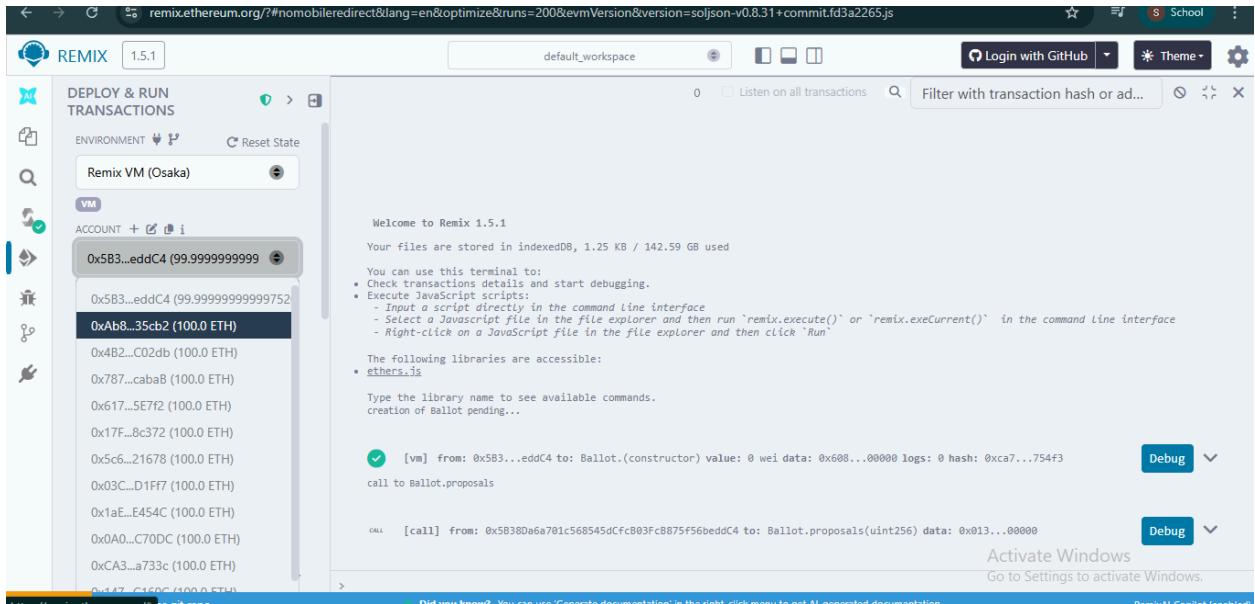
The screenshot shows the Remix IDE interface. On the left, the sidebar displays the deployed contracts: BALLOT AT 0xd91...39138 (ME). The main panel shows the Ballot contract code with the 'giveRightToVote' function highlighted. The terminal window on the right displays the deployment logs and a welcome message for Remix 1.5.1.

Viewing the details of the Proposal Candidate and Giving the right to an account other than and by the chairman

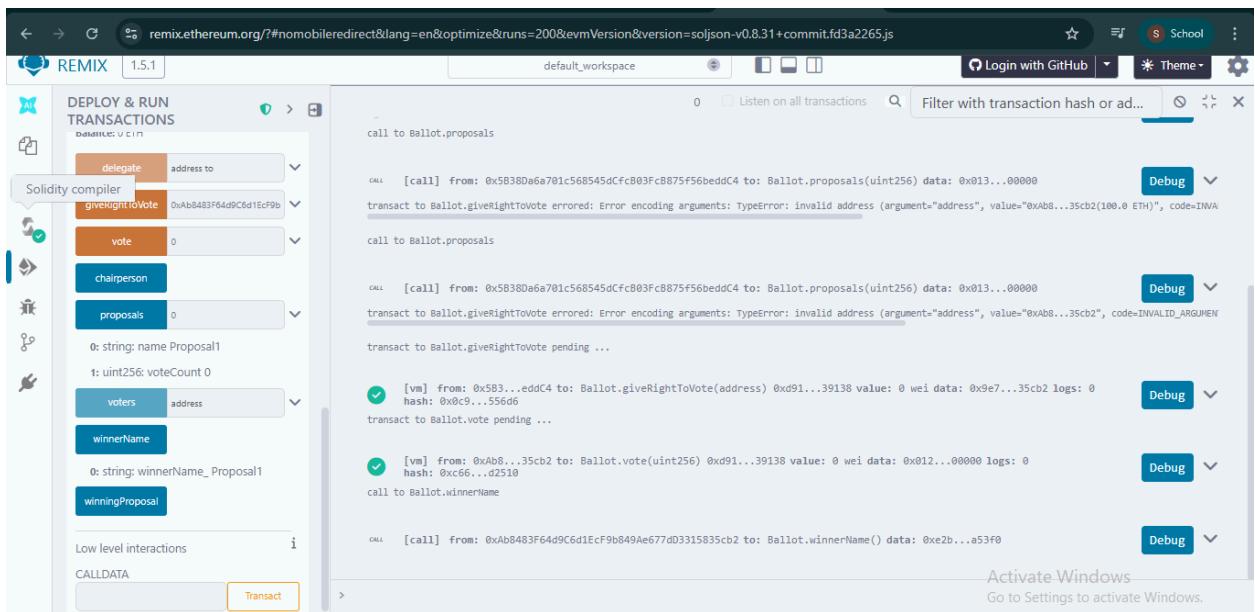


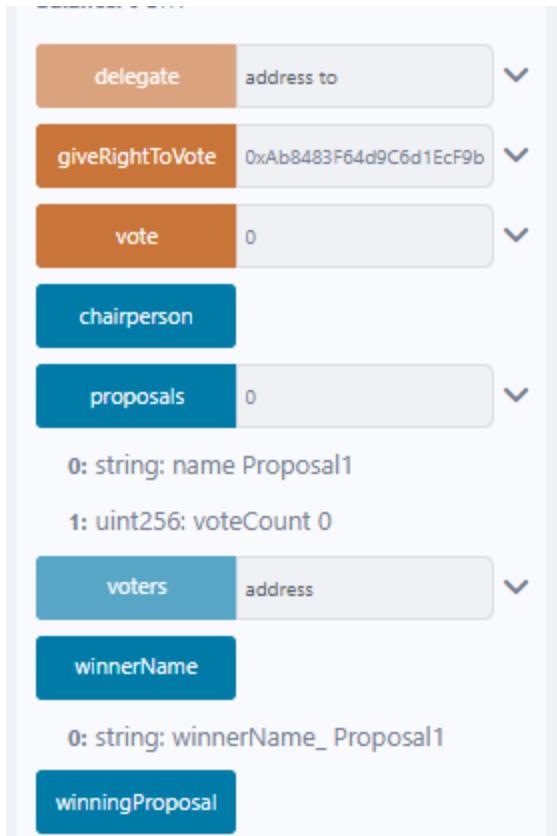
The screenshot shows the Remix IDE interface. The 'giveRightToVote' function is selected in the interface. The terminal window on the right shows the transaction logs for calling this function, indicating it was successful.

Selecting the account which was given the right to vote and then writing the proposal candidate's index to vote.



In this final screenshot we can see that after the delegation's vote the weight of one vote of the one voter who was delegated is the number of people who delegated the voter as their voter (Default weight of any voter is 1).





CONCLUSION:-

In this practical assignment, a Voting (Ballot) Smart Contract was implemented and deployed using Solidity through the Remix IDE environment. The experiment enabled a deeper understanding of how decentralized applications function on the Ethereum blockchain. Essential features such as `require` statements, `mapping`, and data location specifiers (`storage` and `memory`) were utilized to maintain validation, proper data management, and reliable execution of the contract.

Additionally, the comparison between `bytes32` and `string` data types demonstrated the balance that developers must maintain between minimizing gas consumption and improving readability.

Overall, this exercise enhanced conceptual clarity as well as practical knowledge of smart contract architecture, highlighting the importance of efficiency, security, and thoughtful design in blockchain-based voting systems.

